Dynamic Load Balancing Based on Applications Global States Monitoring

Eryk Laskowski, Marek Tudruj, Richard Olejnik, Damian Kopanski

Dynamic Load Balancing Based on Applications Global States Monitoring

Eryk Laskowski*, Marek Tudruj*†, Richard Olejnik‡ and Damian Kopański†
*Institute of Computer Science PAS, 01-248 Warsaw, Jana Kazimierza 5, Poland
Email: {laskowsk,tudruj}@ipipan.waw.pl
†Polish-Japanese Institute of Information Technology, ul. Koszykowa 86, 02-008 Warsaw, Poland
Email: {damian.kopanski,tudruj}@pjwstk.edu.pl
‡Computer Science Laboratory of Lille (UMR CNRS 8022), University of Sciences and Technologies of Lille, France
Email: Richard.Olejnik@lifl.fr

***Abstract*—The paper presents how to use a special novel distributed program design framework with evolved global control mechanisms to assure processor load balancing during execution of application programs. The new framework supports a programmer with an API and GUI for automated graphical design of program execution control based on global application states monitoring. The framework provides high-level distributed control primitives at process level and a special control infrastructure for global asynchronous execution control at thread level. Both kinds of control assume observations of current multicore processor performance and communication throughput enabled in the executive distributed system. Methods for designing processor load balancing control based on a system of program and system properties metrics and computational data migration between application executive processes is presented and assessed by experiments with execution of graph representations of distributed programs.**

Keywords*—*distributed programming paradigms;global application states monitoring;graphical program design tools.

I. Introduction

Load balancing is the fundamental approach used to optimize execution time of distributed programs. In a multi-user environment, the availability of computing resources can vary notably over time. Thus, an optimization subsystem, embedded in the run-time environment or in distributed applications is essential. Since the problem of load balancing of computational tasks is NP–hard, heuristics from various fields have been applied, ranging from prefix sum, recursive bisection, space filling curves to work stealing and graph partitioning. Good reviews of load balancing methods have been presented in [15] — [17].

Static load balancing problems when the computational tasks co-exist during the entire execution of parallel programs can be modeled as graph partitioning problems. METIS [8] is the most popular example of a graph partitioning framework, which has been used for mapping and static load balancing of parallel applications.

In the case of dynamic load balancing formulation with on–line load balancing in the presence of variations of a workload and the varying availability of resources, there exist several load balancing strategies [9]. Dynamic load balancing can be implemented by a migration of the application components (processes and threads) or by a data redistribution among computing nodes that guarantee a possibly high efficiency of the overall application execution. The simplest dynamic load balancing methods are based on the greedy heuristics, where the largest workloads are moved to the least loaded processors until the load of all processors is close to the average load. More sophisticated algorithms use some refinement strategies, where the number of migrated objects is reduced or the communication between different objects is also considered.

Monitoring of global application states [1] creates an efficient and flexible basis for distributed program execution control. Unfortunately, no existing parallel run-time system provides a built-in infrastructure for these purposes. This has been the motivation for our research on a new distributed program design framework called PEGASUS (from Program Execution Governed by Asynchronous SUpervision of States) [4] which is assumed in this paper as a basic program control infrastructure.

In the PEGASUS framework the semantics of program execution control constructs at process and thread levels takes into account automatic monitoring of global application states. The proposed methods include new program execution paradigms and the corresponding software architectural solutions. The global flow control constructs assume modular structures of parallel programs based on the notions of processes and threads. The global control constructs logically bind program modules and define the involved control flow selectively dependent on global application states. The internal behaviour of processes and threads is also asynchronously controlled by inspecting global application states. The PEGASUS control infrastructure provides synchronizers which collect local state information from processes and threads, automatically construct global strongly consistent application states, evaluate relevant control predicates on global states and provide a distributed support for sending control Unix-type signals to distributed processes and threads to stimulate the desired control reactions. The repertoire of considered local and global states, the control predicates and the reactions to them are user programmed using a special API provided in the system.

The design of the program global execution control is graphically supported and decoupled from data processing control inside process and thread modules. The proposed global control constructs enable better verification and are less error prone.

The contribution of this paper is a general load balancing method based on the special infrastructure for monitoring of application program global states and runtime executive system behavior observation. We are interested in dynamic load balancing, where load distribution can be changed during execution, following variations in system resources availability and/or changes of their computational load. In the strategy presented in the paper we focus at data migration as a basic load balancing mechanism. The presented approach leverages some earlier works, reported in [2], [4]. The use of load balancing methods based on graph partitioning like in existing load balancing libraries (METIS [8], Zoltan [5], etc.) is also possible under PEGASUS. The complete graph partitioning algorithms can be embedded in the PEGASUS control infrastructure which is fully programmable. However, it would require an extensive load redistribution by numerous time-consuming distant processor thread-to-thread load transfers, to follow the global optimal work partition. In our strategy, we propose to avoid such strategy and to replace it by workload distribution control, however, allowing for real load migration if unavoidable by other methods.

When we analyze features of current parallel computing environments like CHARM++ [6] for C++ or ProActive [7] for Java, we notice the absence of any automatic infrastructure offered to a programmer to support the monitoring and using of global parallel application states in the design of program execution control. Both CHARM++ and ProActive introduce their own high-level parallel programming paradigms. CHARM++ programs are composed of message-driven objects called *chares*, while ProActive is based on the *active object* model. PEGASUS, at the parallel program implementation level, supports parallel programming modularity based on processes and thread blocks. PEGASUS offers a unique program execution control design infrastructure based on a global application states monitoring. MPI is used for message passing based global data communication at the thread and process levels and the OpenMP/Pthreads are used for internal process parallelism and data communication at the thread level. The PEGASUS control infrastructure and underlying system architectural concepts are used to organize dynamic load balancing control in distributed applications.

The rest of the paper consists of three parts. In part II the proposed program execution model is described. Part III describes the proposed load balancing strategy, implemented using global application states monitoring. Part IV describes the experimental assessment of the presented algorithm.

II. Application execution model

Distributed application programs based on the new control approach are composed of processes and threads inside each process. An *executive system* consists of multicore processor nodes interconnected by a message passing network (e.g. a cluster of workstations). The network is used for data communication at the process level. A programmer is able to control assignments of processes to processor nodes. We assume that processor nodes work under control of Linux operating system.

The PEGASUS runtime environment provides an application program designer with a *control infrastructure* which enables organizing program execution control based on the global application states. It uses a number of graphical and communication library mechanisms provided as a parallel program design infrastructure.

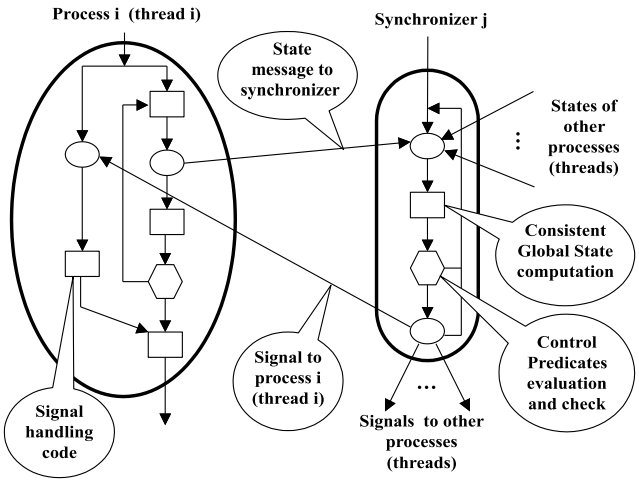


Fig. 1. Processes (threads) co-operating with a synchronizer.

Application program execution control is organized at two layers:
(1) a global control layer, responsible for monitoring global execution states of processes or threads in application programs, computing control predicates on global states and issuing signals to application processes or threads, to stimulate desired reactions,
(2) a local control layer, which is responsible for reporting by processes and threads their local states to the global control layer as well as for organizing reactions to control signals coming from the global control layer.

Monitoring of application program global states will influence program execution by acting on control inside distributed application processes or threads to asynchronously modify their behavior in a manner similar to distributed interrupts issued based on global application states monitoring, or on application synchronous global control flow influenced by monitoring of global application states.

A. Asynchronous program execution control

The general scheme of the proposed control mechanism is shown in Fig. 1. Application program processes (threads) send messages on their *states* to special globally accessible processes (threads) called *synchronizers*. A synchronizer collects local state messages, determines if the application's strongly consistent global state has been reached, evaluates control predicates and stimulates desired reactions to them in application components.

A strongly consistent global state (SCGS) means a set of fully concurrent local states detected unambiguously by a synchronizer [1]. Processor node clocks are synchronized with a known accuracy to enable the construction of strongly consistent global states by projecting the local states of all processes or threads on a common time axis and finding time intervals which are covered by local states in all participating processes or threads [3].

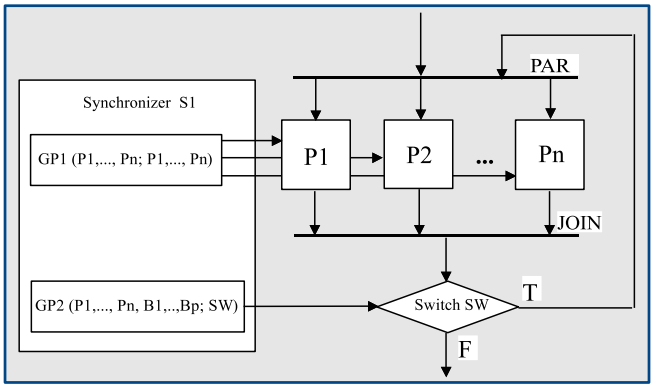


Fig. 2. PARALLEL DO-UNTIL construct with an Asynchronous Predicate GP1 and a Control Flow predicate GP2.

On each global state reached, one or more control conditions (control predicates) are computed. Predicates are specified as blocks of code in C. If a predicate value is true, then a number of control signals are sent by the synchronizer to selected application processes (threads). In the code of a process (thread), regions sensitive to incoming control signals can be marked by special delimiters. If the process (thread) control is inside a region sensitive to a signal and the signal arrives, then a reaction is triggered. Otherwise the reaction is neglected. Two types of reaction to control signals are provided:
— signal-driven activation (interrupt), which breaks current computation and activates a reaction code associated with the region. After completion of the reaction code the broken computing resumes,
— signal-driven cancellation, which stops computation and activates a cancellation handling procedure associated with the region. Program execution resumes just after the abandoned region.

B. Control flow governed by global states monitoring

The second control mechanism involving the global state monitoring concerns defining the flow of control in distributed programs based on the global application state monitoring. The global parallel control structures provided in the system are based on PARALLEL DO (PAR) and JOIN constructs, embedded (if needed) into standard control statements of high level languages (IF, WHILE...DO, DO...UNTIL, CASE) but governed by predicates on application global states. Fig. 2 shows a global PARALLEL DO–UNTIL control construct. The predicate *GP1* asynchronously controls execution of program blocks *P1, ..., Pn*. It receives local state messages from *P1, ..., Pn*, evaluates a predicate condition and sends control signals to *P1, ..., Pn*. The predicate *GP2* receives local state messages from program blocks *P1, ..., Pn*, *B1, ..., Bp* and sends a binary control signal to switch *SW*, which governs the flow of control in the PARALLEL DO–UNTIL construct.

The distributed Execution Control (EC) process in the system coordinates program execution and manages code blocks (process) creation and activation resulting from global parallel control constructs. More details on the PEGASUS framework can be found in [4].

III. Load balancing based on global states

A. Load balancing algorithm

The global state monitoring infrastructure of the PEGASUS environment is used as a tool to implement dynamic load balancing at the application level. The computing nodes (workstations) can be heterogeneous, moreover they can have different and variable computing capabilities over time. A load imbalance occurs when the differences of workload between the computing nodes become too big. We distinguish two main steps in load balancing: detection of imbalance and its correction. The first step uses measurement tools to detect the functional state of the system. The second consists in migrating some load from overloaded computing nodes to underloaded computing nodes to balance the workloads.

An intrinsic element of load balancing is the application observation mechanism. It provides knowledge of the application behavior during its execution. This knowledge is necessary to undertake adequate and optimal load balancing decisions. There are two types of measurements in the proposed load balancing method for PEGASUS environment:

– *system level observations*, which provide general functional indicators, e.g. CPU load, that are universal for all kinds of applications; the system measurements are implemented using the software agent approach, i.e. the load balancing mechanism being part of PEGASUS environments deploys observation agents on computing nodes.

– *application specific observations*, which incorporate measurements that have to be implemented in each application, as they provide information about application–dependent behavior; an example of this kind of indicator is the workload of a process (or thread) since it can depend on the volume of data to be processed in the future which is known only to application logic.

The aforementioned observation mechanisms are organized using the PEGASUS global execution states monitoring infrastructure. Application program processes and system observation agents send messages on local state changes to *load balancing synchronizer*, where they are processed and appropriate reactions are computed using the method described in next sections. Similarly, reactions are organized as asynchronous program execution control. Load balancing logic is implemented as control predicates inside *a load balancing synchronizer*. Figure Alg. 1 presents a general scheme of the proposed algorithm. The rest of this section describes functions and symbols used in the pseudo-code in Algorithm 1.

B. Detection of load imbalance

To detect load imbalance, the knowledge on the functional state of computing nodes composing the cluster is essential. As the environment is heterogeneous, it is necessary to know not only the load of computing nodes but also their computing power capabilities. The heterogeneity disallows us to directly compare measurements based on program execution time taken on computing nodes whose computing powers are different. After experiments to determine the computing node power, we have found that the parameter, which allows us to compare the computing nodes' load is the availability index of a CPU

Algorithm 1 General scheme of load balancing algorithm

initialize *load balancing synchronizer*
loop
{Global part of the algorithm}
wait for state change
store values $\mathrm{Ind}_{\mathrm{avail}}$
$LI \leftarrow \max_{n\in N}(\mathrm{Ind}_{\mathrm{avail}}(n)) \geq \alpha * \min_{n\in N}(\mathrm{Ind}_{\mathrm{avail}}(n))$
if LI **then**
{Step 1: Classification of the computing nodes' load}
$N_U, N_N, N_O \leftarrow$ classify_nodes_load {K–Means algorithm, $K = 3$: underloaded, normal, overloaded}
{Local part of the algorithm}
for all $n \in N_O$ **do** {in parallel}
{Step 2: Choice of candidates for migration}
$rank_{\min} \leftarrow \infty$
for $j \in T(n)$ **do**
$\mathrm{Rank}(j) \leftarrow \beta * \mathrm{attr}^{\%}(j) + (1-\beta) * \mathrm{ldev}^{\%}(j)$
if $\mathrm{Rank}(j) < rank_{\min}$ **then**
$rank_{\min} \leftarrow \mathrm{Rank}(j)$
$j_n \leftarrow j$ {candidate for migration}
end if
end for
end for
{Step 3: Selection of migration target}
for $u \in N_U$ **do**
$qual_{\max} \leftarrow 0$
for $n \in N_O$ **do**
$\mathrm{Quality}(j_n, u) \leftarrow \gamma * \mathrm{attrext}^{\%}(j_n, u) + (1-\gamma) * \mathrm{Ind}^{\%}_{\mathrm{avail}}(u)$
if $\mathrm{Quality}(j_n, u) > qual_{\max}$ **then**
$qual_{\max} \leftarrow \mathrm{Quality}(j_n, u)$
$target(j_n) \leftarrow u$ {target of migration}
end if
end for
send_signal $migrate(j_n \Rightarrow target(j_n))$
$N_O \leftarrow N_O - \{n\}$
end for
end if
end loop

computing power on the node n:

$$\mathrm{Ind}_{\mathrm{avail}}(n) = \mathrm{Ind}_{\mathrm{power}}(n) * \mathrm{Time}^{\%}_{\mathrm{CPU}}(n)$$

where:
$\mathrm{Ind}_{\mathrm{power}}(n)$ — computing power of a node n, which is the sum of computing powers of all cores on the node,
$\mathrm{Time}^{\%}_{\mathrm{CPU}}(n)$ — the percentage of the CPU power available for programs under load balancing on the node n, periodically estimated by observation agents on computing nodes.

Some explanation is needed to clarify the way the availability index of a CPU computing power is computed. The computing power of the node is the outcome of the *calibration process* [10]. For each node, the calibration should be performed in a consistent way to enable comparisons of calibration results (they can be expressed in MIPS, MFLOPS, Dhrystones or similar). The calibration needs to be done only once when the nodes join the system. The percentage of the CPU power available for a single computing thread is computed as a quotient of the time during which the CPU was allocated to the probe thread against the time span of the measurement (see [10] for details and the description of the implementation technique). $\mathrm{Time}^{\%}_{\mathrm{CPU}}(n)$ value is the sum of the percentage of CPU power available for the number of probe threads equal to the number of CPU cores in the node.

A load imbalance LI is defined based on the difference of the availability indices between the most heavily and the weighted least heavily loaded computing nodes composing the cluster, which can be determined as:

$$LI = \begin{cases} true & \text{if} \quad \max_{n\in N}(\mathrm{Ind}_{\mathrm{avail}}(n)) - \\ & \quad \alpha * \min_{n\in N}(\mathrm{Ind}_{\mathrm{avail}}(n)) \geq 0 \\ false & \text{otherwise} \end{cases}$$

where:
N — the set of all computing nodes,
α — a positive constant number.

Power indications $\mathrm{Ind}_{\mathrm{power}}$ and CPU time use rate $\mathrm{Time}_{\mathrm{CPU}}$ are collected and sent to *load balance synchronizer* by local system agents as state messages.

The proper value of the α coefficient can be determined using both statistical and experimental approaches. Following our previous research [11] on load balancing algorithms for Java-based distributed computing environment, we can restrict the value to the interval $[1.5 \ldots 2.5]$. It enables controlling the sensitivity (and frequency) of detection of load imbalance for small differences in computing power availability in homogeneous systems and for heterogeneous processor clusters in the case of very fast and slow CPUs appearing in the system.

C. Correction of load imbalance

In this step we detect overloaded computing nodes and then we transform them into the normally loaded (rebalance).

1) Classification of computing nodes: We use the K-Means algorithm [12] to build categories of computing nodes based on the computed availability indices. We classify n computing nodes into the $K = 3$ categories: underloaded (N_U), normally loaded (N_N) and overloaded (N_O). The three centers of these categories are values of availability indices close to the minimum, average and maximum over the whole cluster of computing nodes.

2) Choice of candidates for migration: The loads are represented by the data processing activities of the threads which are running on computing nodes. To correct load imbalance, we have to migrate the load from overloaded computing nodes to underloaded ones. Two parameters are used to find the load that we want to migrate:

a) the *attraction* of a load to a computing node,
b) the *weight* of the load.

The attraction of a load to a computing node is expressed in terms of communication, i.e. it indicates how much a particular thread communicates with others allocated to the same node. A strong attraction means frequent communication, so, the less the load is attracted by the current computing node, the more interesting it is to be selected as a migration candidate. The computational weight of the load gives the quantity of load which could be removed from the current node and placed on another.

Both the *attraction* and *weight* are application-specific metrics, which should be provided by an application programmer in the form of state messages sent to *load balance synchronizer*:

1) $\mathrm{COM}(t_s, t_d)$ is the communication metrics between threads t_s and t_d,
2) $\mathrm{WP}(t)$ is the load weight metrics of a thread t.

$\mathrm{WP}(t)$ can be any measure of a thread work, for example the number of instructions to be executed in a thread. Our strategy for selecting threads for migration is to promote threads which show loads with a small distance to the average thread load, not to involve dramatic load changes after a single thread migration.

The attraction of the load j to the actual computing node is defined as:

$$\mathrm{attr}(j) = \sum_{o \in L^*(j)} (\mathrm{COM}(j,o) + \mathrm{COM}(o,j))$$

where:
$L^*(j)$ — the set of threads, placed on the same node as a thread j (excluding j).

The load deviation compared to the average quantity of work of the node j is defined as:

$$\mathrm{ldev}(j) = |\mathrm{WP}(j) - m_{WP}| \tag{1}$$

where:
$m_{WP} = \frac{\sum_{o \in L(j)} \mathrm{WP}(o)}{|L(j)|}$,
$L(j)$ — the set of threads, placed on the same node as the thread j (including j).

The element to migrate is the one for which a weighted sum of the normalized attraction and load deviation has the minimal value:

$$\mathrm{Rank}(j) = \beta * \mathrm{attr}^{\%}(j) + (1-\beta) * \mathrm{ldev}^{\%}(j) \tag{2}$$

where:

$$\mathrm{attr}^{\%}(j) = \frac{\mathrm{attr}(j)}{\max_{o \in L(j)}(\mathrm{attr}(o))}$$

$$\mathrm{ldev}^{\%}(j) = \frac{\mathrm{ldev}(j)}{\max_{o \in L(j)}(\mathrm{ldev}(o))}$$

β — a real between 0 and 1. Its choice remains experimental. Let us notice however that the bigger β is, the bigger is the weight of the object attraction.

3) Selection of the target computing node for migration: The first criterion to qualify a computing node as a migration target is the attraction of a selected load entity to this node. The attraction of the load j to node n is defined as follows:

$$\mathrm{attrext}(j,n) = \sum_{e \in T(n)} (\mathrm{COM}(e,j) + \mathrm{COM}(j,e))$$

where:
$T(n)$ — the set of threads, placed on a node n.

The second criterion is based on the computing node power availability indices. We prefer the one whose availability index is the highest, because it is actually the least loaded. We also take into account the number of *waiting* threads in the potential targets ($T_{\mathrm{wait}}(n)$ – the set of waiting threads on a node n). We consider them, however, as potential load, which must be taken under consideration with the related load currently done on the machine. The formula to select the target for migration is as follows (we normalize all the values related in the interval $[0 \ldots 1]$):

$$\mathrm{Quality}(j,n) = \gamma * \mathrm{attrext}^{\%}(j,n) + (1-\gamma) * \mathrm{Ind}^{\%}_{\mathrm{avail}}(n) \tag{3}$$

with $\gamma \in [0 \ldots 1]$ and

$$\mathrm{attrext}^{\%}(j,n) = \frac{\mathrm{attrext}(j,n)}{\max_{e \in N}(\mathrm{attrext}(j,e))}$$

$$\mathrm{Ind}^{\%}_{\mathrm{avail}}(n) = \frac{\mathrm{Ind}^{*}_{\mathrm{avail}}(n)}{\max_{e \in N}(\mathrm{Ind}^{*}_{\mathrm{avail}}(e))}$$

$$\mathrm{Ind}^{*}_{\mathrm{avail}}(n) = \mathrm{Ind}_{\mathrm{avail}}(n) - \mathrm{Ind}_{\mathrm{avail}}(n) * \frac{|T_{\mathrm{wait}}(n)|}{|T(n)|} \tag{4}$$

For a load which is a candidate for migration, we evaluate the above equations for all potential migration targets (underloaded computing nodes). The computing node which maximizes equation 3 will be chosen as the new location for the migrated load. The value of the coefficient γ has to be determined using experimental verification.

D. Implementation under PEGASUS

We will illustrate now implementation of the described load balancing algorithm under PEGASUS framework, taking as an example a simple iterative application, consisting of processes $P1 ... Pn$, run in parallel inside a single DO-UNTIL loop. The control flow graph of the application, including the load balancing infrastructure, is shown in Fig. 3.

The execution of load balancing is globally controlled by the synchronizer *LB*, assigned to a processor of the system. Each application process Pi is composed of a number of application threads Tj and a thread synchronizer *Th*. The thread synchronizer cooperates with the application threads and the global synchronizer *LB*. Th evaluates control predicates on local states transferred from application process threads. Based on the evaluated predicates, some control signals can be sent back to threads and/or some process states can be sent from process synchronizers *Th* to the global synchronizer *LB*. *LB* evaluates some global predicates related to the global load balancing control. Based on these predicates, *LB* sends control signals to synchronizers *Th* in application processes. The synchronizers *Th* can process the received control signals and send the respective signals down to the threads which they supervise.

The progress of iterations in the application is controlled by the *MoreIteration* predicate in the global synchronizer *LB*. The *MoreIteration* predicate receives local states of threads via local synchronizers Th. This global predicate controls the switch *SW*, which directs the flow of control accordingly to the evaluated *MoreIteration* predicate value.

We will now explain the way in which the load balancing algorithm is implemented with the use of the infrastructure of synchronizers and local state/signal communication. Each thread synchronizer *Th* contains an Observation Agent, which

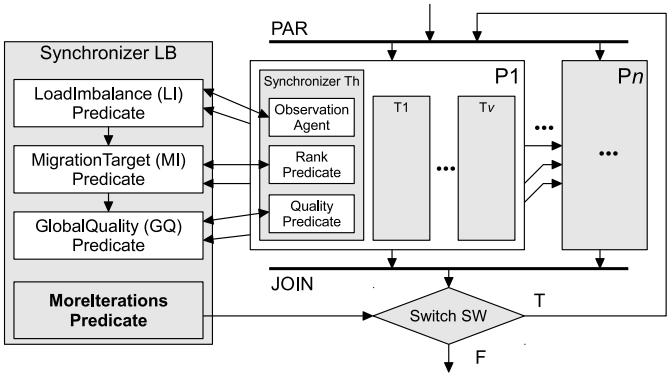


Fig. 3. Control graph of an application based on parallel DO-UNTIL construct under PEGASUS.

periodically evaluates the availability of computing power in processor nodes and sends to the global synchronizer *LB* the local node state message which corresponds to the given node CPU computing power availability index (see $\mathrm{Ind}_{\mathrm{avail}}(n)$ in section III B). *LB* periodically receives such state messages and based on them evaluates the load imbalance predicate (*LI*) in the system (following the *LI* formula shown in section III B). If *LI* = *true*, the K-Means algorithm is activated which classifies the nodes in the system as underloaded, normally loaded and overloaded. The synchronizers *Th* in the overloaded nodes are notified by control signals. As a reaction, the overloaded node *Th* synchronizers activate computing the Rank Predicates to evaluate ranks of all their threads in respect to their eligibility for load migration (see formulas for $\mathrm{Rank}(j)$ in section II C and the Algorithm 1). Next, each overloaded node *Th* select the thread with the minimal rank and sends the identifier of minimal rank thread to the global synchronizer *LB* as the best candidate for load migration from the node.

Based on best candidate messages from the overloaded *Th*s, the global synchronizer LB broadcasts the list of all migration candidate threads as control signals to thread synchronizers Th in all underloaded nodes. The migration candidate lists activate the Quality Predicates in *Th* synchronizers which start evaluating the quality of potentially migrated load placement on underloaded target nodes. It is done based on the reply messages to current load and communication states requests sent by *Th*s to the threads they supervise. In response to the state messages, the Quality Predicates are evaluated in *Th*s (see formula (3) in section III C part 3) for all migration candidates (threads from overloaded nodes). The node for which a candidate thread maximizes the quality of target thread placement is selected as the effective target for the thread placement and its identifier is sent by the *Th* to the Global Quality Predicate in *LB* synchronizer. This predicate selects pairs of overloaded/underloaded nodes for wich the quality of migration is the best. As a result *LB* sends control signals to the *Th* synchronizers of the selected overloaded and underloaded nodes to stimulate them to activate execution of the reduction and increase of loads in processes they supervise.

The load changes are done by cancellation of some workload in the overloaded nodes and reception of this workload for execution in the underloaded nodes. Such load balancing in pairs of nodes is done until all underloaded nodes have been used. Then, the load balancing algorithm returns to the waiting for a new global load balance change which is checked by the evaluation of the *LoadImbalance* predicate in the *LB* synchronizer based on state messages sent by the Observation Agents in application processes in executive system nodes.

IV. Experimental assessment of the presented load balancing algorithm

We will present now an experimental assessment of the presented load balancing algorithm. The experimental results were collected by simulated execution of application programs in a distributed system. The simulated model of execution corresponded to typical message-passing parallel applications using the MPI library for communication. The simulator was based on the DEVS discrete event system approach [13].

The application model used was similar to the model presented in [14] (Temporal Flow Graph, TFG). The application consisted of indivisible tasks (these are threads or processes of the operating system). Each task consisted of several computational blocks, separated by communication (messaging) with other tasks.

Applications run in a cluster of computing nodes. The system consisted of multi-core processors, each of which had its own main memory and a data network interface. Communication contention was modeled at the level of the network interface of each computing node.

During simulation, in parallel with the execution of the application, a dynamic load-balancing algorithm was performed. The algorithm used was the same as presented in the paper, see Algorithm 1. Computing nodes were periodically reporting their loads to the global load balancing synchronizer and then, depending on the states of the system and the application, appropriate actions were undertaken.

During experiments we used a set of 10 exemplary application programs, containing from 16 to 80 tasks. These programs were randomly generated, but their general structure corresponds to layered MPI-based parallel applications which correspond to numerical analysis or physical phenomena simulation. Each application program consisted of a set of phases. Each phase consisted of a fixed number of computational tasks

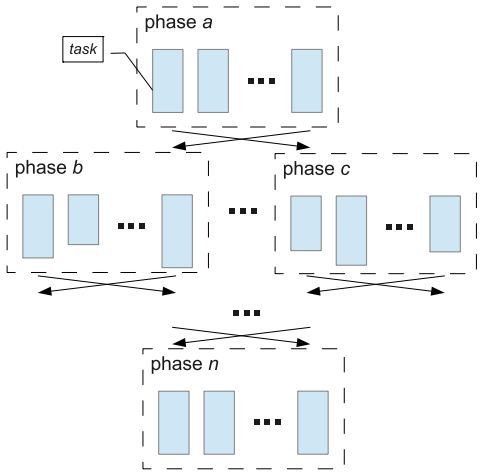
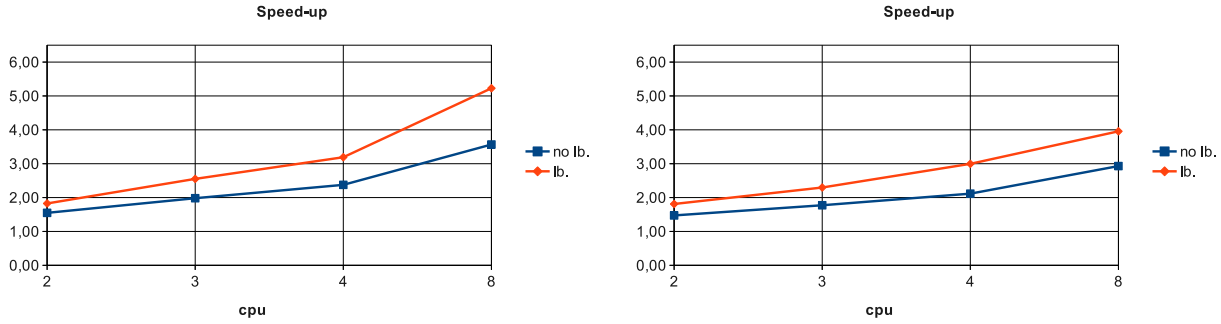


Fig. 4. The general structure of exemplary applications.



(a) (b)

Fig. 5. Speed-up for different number of nodes with or without load balancing (a) irregular applications. (b) regular applications.

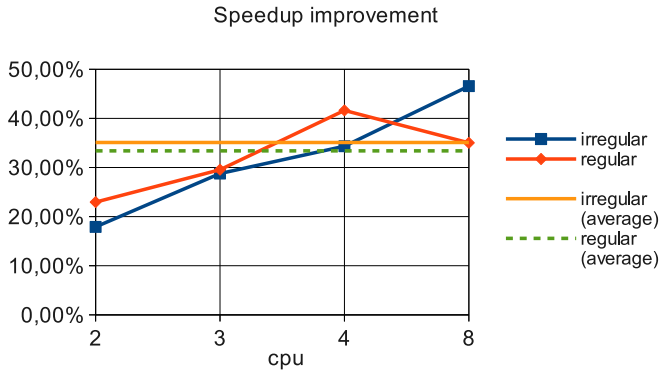


Fig. 6. Speedup improvement for irregular and regular applications: the average and for different number of nodes in the system (METIS-based initial task placement).

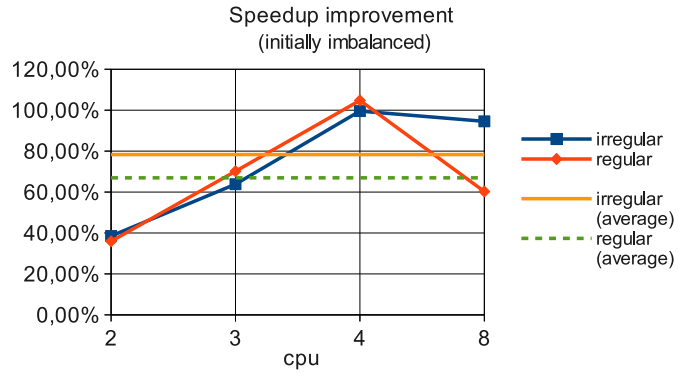


Fig. 7. Speedup improvement for irregular and regular applications for unoptimized initial placement of application tasks.

(i.e. threads), Fig 4. Tasks communicated with other tasks of the same phase. On the boundaries between phases, there was a global exchange of data. Phases could be executed sequentially or in parallel (depending on the particular exemplary application). The difference between regular and irregular applications is that the execution time of tasks in some (or all) phases of irregular applications depends on the processed data. From the outside, irregular applications exhibit the behavior in which the execution time of tasks and the communication scheme seem unpredictable. Thus, a load imbalance can occur in computing nodes.

We simulated execution of applications in systems with 2, 3, 4 or 8 identical computing nodes, each containing the same number of cores. Since the execution times of the same application for different runs can vary, the results are the averages of 5 runs of each application.

The summary of the average speed-up improvement resulting from load balancing performed by the algorithm presented in the paper is shown in Fig. 6. The average speed-up improvement over the execution without load balancing is big for both irregular and regular applications. This could be considered as an unexpected result, since the good initial placement of tasks of regular applications usually could be statically calculated before application execution. The reason for the improvement is that the initial placement of tasks was far from optimal for both categories of applications, even when we used the METIS graph partitioning algorithm for its calculation. There are usually intense data dependencies between phases of applications, so the program execution scenario was effectvely much improved by dynamic load balancing.

On Fig. 5(a) and 5(b) the speed-up of irregular and regular applications for different number of computing nodes is shown. Our exemplary regular applications give smaller speed-up than irregular ones (with or without load balancing).

In the case of the totally unoptimized initial placement of applications tasks, the dynamic load balancing algorithm gives the big speedup improvement, Fig. 7.

During experimests, we measured the cost of dynamic load balancing as the number of tasks migrations during the execution of applications, Fig. 8. The cost depends mainly on the quality of the initial placement of tasks and the category of an application. For poor (i.e. unoptimized) initial placement of tasks, the cost of dynamic load balancing is much higher. Moreover, the irregular applications require more frequent load balancing at the run-time, resulting from the unpredictable communication and computation scheme of their execution. However, even for the optimized initial placement of application tasks there is a need for dynamic load balancing at run-time when external execution conditions are changing (e.g. varying load of computing nodes dependent on other applications or operating system activity).

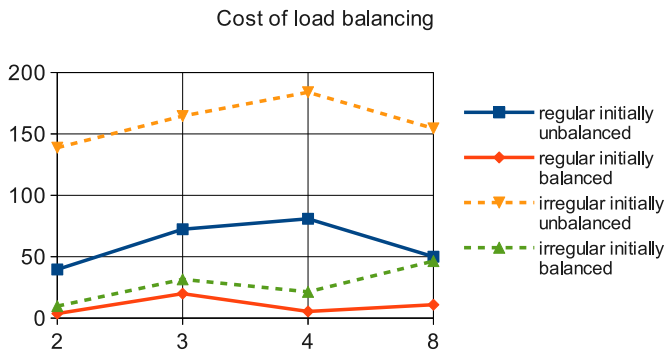


Fig. 8. The cost of dynamic load balancing shown as the number of migrations per single application execution.

The proposed load balancing algorithm is meant for the PEGASUS environment based on global application states monitoring. The algorithm was implemented in the frame of the cooperation of application tasks with synchronizers. The tasks send load reports to synchronizers and modify their behavior in response to signals received asynchronously from the synchronizers.

V. Conclusions

Dynamic load balancing in distributed systems based on application global states monitoring has been discussed in this paper. Global control constructs for the control flow design and asynchronous state monitoring with control signal dispatching provide flexible infrastructure for application implementation and system-level optimizations. The infrastructure enables an easy and convenient design of the load balancing logic in applications. Our experimental results collected so far confirm that the presented load balancing method performs well for different run-time conditions.

The proposed PEGASUS framework is currently in the final implementation stage for a multicore processors cluster interconnected by dedicated separate networks for control and computational data communication as well as for processor clock synchronization for strongly consistent global states discovery. Inter-process control communication including Unix signal propagation between processors is organized by the use of message passing over Infiniband network. Computational data communication between processors is performed by an Ethernet network. C/C++ language with the MPI2, OpenMP and Pthreads libraries are used for writing application programs and the framework control code.

Important features of the load balancing implemented under PEGASUS algorithm are the expected low overheads, the ease in programming and tuning the load balancing algorithms as well as the ability to organize load balancing in distributed manner due to the ready-to-use infrastructure of asynchronous control based on global application states monitoring. The communication overhead of the load balancing algorithms can be strongly reduced due to the use, in the system infrastructure, of totally separate program layers and separate physical networks for control communication and data communication in applications. Activities of load balancing actions can be almost completely overlapped with application computations due to the assumed asynchronous type of control and the possible use of dedicated resources for load balancing control computations (the use of separate synchronizer threads assigned to separate processor cores).

An interesting topic of further research, which we were unable so far to cover, is the periodic use of the METIS algorithm to support load balancing by program graph partitioning, which can be easily embedded inside synchronizers activities.

This research was partially supported by the national MNiSW research grant No. NN 516 367 536.

References

[1] O. Babaoglu, K. Marzullo, Consistent global states of distributed systems: fundamental concepts and mechanisms, Distributed Systems, Addison-Wesley, 1995.

[2] M. Tudruj, J. Borkowski, D. Kopański, Load balancing with migration based on synchronizers in PS-GRADE graphical tool. Int. Symp. on Parallel and Distributed Computing, ISPDC 2005, Lille, France, July 2005, IEEE CS, pp. 105–112

[3] S. D. Stoller, Detecting Global Predicates in Distributed Systems with Clocks, Distributed Computing, Vol. 13, N. 2, 2000, pp. 85-98.

[4] M. Tudruj, J. Borkowski, Ł. Maśko, A. Smyk, D. Kopański, E. Laskowski, Program Design Environment for Multicore Processor Systems with Program Execution Controlled by Global States Monitoring. 10th Internat. Symposium on Parallel and Distributed Computing, Cluj-Napoca, July, 2011, ISPDC 2011, IEEE CS, pp. 102-109.

[5] K.D. Devine, E.G. Boman, L.A. Riesen, U.V. Catalyurek, C. Chevalier, Getting Started with Zoltan: A Short Tutorial, Sandia Nat. Labs Tech Rep. SAND2009-0578C, 2009.

[6] L.V. Kalé, S. Krishnan, CHARM++: A Portable Concurrent Object Oriented System Based on C++, Proc. of OOPSLA'93, ACM Press, Sept. 1993, pp. 91–108.

[7] Baude et al., Programming, Composing, Deploying for the Grid, GRID COMPUTING: Software Environments and Tools, J. C. Cunha, O. F. Rana (Eds), Springer, Jan. 2006.

[8] G. Karypis, V. Kumar, Multilevel graph partitioning schemes, Proc. 24th Intern. Conf. Par. Proc., III. CRC Press, 1995, pp. 113–122.

[9] A. Bhatele, S. Fourestier, H. Menon, L. V. Kale, F. Pellegrini, Applying graph partitioning methods in measurement-based dynamic load balancing, PPL Technical Report 2012, University of Illinois, Dept. of Computer Science.

[10] G. Paroux, B. Toursel, R. Olejnik, V. Felea, A Java CPU calibration tool for load balancing in distributed applications, IS-PDC/HeteroPar'04, IEEE CS 2004, pp. 155–159.

[11] R. Olejnik, I. Alshabani, B. Toursel, E. Laskowski, M. Tudruj, Load Balancing Metrics for the SOAJA Framework, Scalable Computing: Practice and Experience, 2009, Vol. 10, No. 4.

[12] J.A. Hartigan, M.A. Wong, A K-Means clustering algorithm, Applied statistics, Vol. 28, pp. 100-108, 1979.

[13] B. Zeigler, Hierarchical, modular discrete-event modelling in an object-oriented environment, Simulation 49 (5), 1987, pp. 219-230.

[14] C. Roig, A. Ripoll, F. Guirado, A New Task Graph Model for Mapping Message Passing Applications, IEEE Trans. on Parallel and Distrib. Systems, Vol. 18 Issue 12, December 2007, pp. 1740–1753.

[15] K.J. Barker, N.P. Chrisochoides, An Evaluation of a Framework for Dynamic Load Balancing of Highly Adaptive and Irregular Parallel Applications, Supercomputing 2003, November 2003.

[16] G. Zheng, E. Meneses, A. Bhatele, L. V. Kale, Hierarchical Load Balancing for Charm++ Applications on Large Supercomputers, 39-th Int. Conf. on Parallel Processing Workshops, 2010, IEEE CS, pp. 436-444.

[17] M. Randles, D. Lamb, A. Taleb-Bendiab, A comparative study into distributed load balancing algorithms for cloud computing, IEEE 24-th Int. Conf. on Advanced Informatin Networking and Applications, 2010, pp. 551-556.